```java
// comparator for ordering nodes in priority queue
// node with higher upper bound comes first in priority queue
    public void getMaxProfit() {
        int[] pickedUpItems = new int[n];
        // sort identifiers based on the the v/w of the items they refer to
        List<Integer> identifierList = Arrays.stream(identifiers).boxed().collect(Collectors.toList());
        @SuppressWarnings("unchecked")
        ArrayList<Integer> sortedIdentifers = new ArrayList(identifierList);
        Collections.sort(sortedIdentifers, (right, left) -> Double.compare(vWRatio[identifierList.indexOf((left))],
                vWRatio[identifierList.indexOf((right))]));

        Comparator<Node> comparator = new NodeComparator();

        // create root node and insert it into the queue
        PriorityQueue<Node> pq = new PriorityQueue<Node>(n, comparator);
        // new node has upper bound = maximum v/w *capacity, and value =0
        pq.add(new Node("", vWRatio[sortedIdentifers.get(0)] * capacity, 0, capacity));

        // explore state space tree until we explore the entire tree
        // or until we prune all the remaining branches
        while (elapsedTime < 1 * 60 * 1000 && !pq.isEmpty()) {
            Node temp = pq.poll();
            if (temp.ub <= maxValue) {
                // all the other paths have upperbounds less than already found solution
                break;
            }

            // checks if a node is the last possible parent in it's branch and finds it's
            // value
            // increase maxValue if applicable
            if (temp.solution.length() == n - 1) {
                // last parent if it can take the last item
                if (temp.remainingCap >= weights[sortedIdentifers.get(temp.solution.length()) - 1]) {
                    int lastVal = values[sortedIdentifers.get(temp.solution.length()) - 1];
                    if (temp.value + lastVal > maxValue) {
                        maxValue = temp.value + lastVal;
                        maxString = temp.solution + "1";
                    }
                }
                // last parent if it can't take the last item
                else {
                    if (temp.value > maxValue) {
                        maxValue = temp.value;
                        maxString = temp.solution + "0";
                    }
                }
            } else {
                // value and weight of next best v/w item
                int lastVal = values[sortedIdentifers.get(temp.solution.length()) - 1];
                int lastWeight = weights[sortedIdentifers.get(temp.solution.length()) - 1];

```

```
52                  // create and enque left and right node, for the next best v/w
item being taken
53                  // or not
54                  // create right node only if the next best item can be picked
up
55                  if (temp.remainingCap >=
weights[sortedIdentifers.get(temp.solution.length()) - 1]) {
56                      Node right; // take the next item
57                      // upper bound is value of next best item + value of
parent + potential value
58                      double rightUB = getUB(temp.solution.length() + 1,
temp.value + lastVal,
59                              temp.remainingCap - lastWeight, sortedIdentifers);
60                      right = new Node(temp.solution + "1", rightUB, temp.value
+ lastVal,
61                              temp.remainingCap - lastWeight);
62                      pq.add(right);
63                  }
64                  Node left; // don't take the next item
65                  // upper bound is value of next best item + potential value
66                  double leftUB = getUB(temp.solution.length() + 1, temp.value,
temp.remainingCap, sortedIdentifers);
67                  left = new Node(temp.solution + "0", leftUB, temp.value,
temp.remainingCap);
68                  pq.add(left);
69              }
70              // update timer
71              elapsedTime = (new Date()).getTime() - startTime;
72          }
73      }
74      // Goes over the remaining best v/w items, and picks them up
75      // if it has the capacity to
76      // and picks up a fraction of them if it can't pick up the entire item
77      public double getUB(int numSeen, int combinedVal, int remainingCap,
ArrayList<Integer> sortedIdentifers) {
78      // upper bound starts at parent value for left and parent value + next
item
79      // value for right
80      double upperBound = combinedVal;
81      int tempCapacity = remainingCap;
82      ListIterator<Integer> li = sortedIdentifers.listIterator(numSeen);
83      while (li.hasNext() && tempCapacity > 0) {
84        int picked = li.next();
85        // try to pick up item whole
86        if (weights[picked - 1] <= tempCapacity) {
87          upperBound += values[picked - 1];
88          tempCapacity -= weights[picked - 1];
89        }
90        // pick up fractional item
91        else {
92          double fractionalUB;
93          fractionalUB = vWRatio[picked - 1] * (tempCapacity);
94          upperBound += fractionalUB;
95          tempCapacity = 0;
96        }
97      }
98      return upperBound;
99    }
```